Programming the Illiac IV

David K. Stevenson

November, 1975

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

This research was supported in part by the Office of Naval Research under contract N0014-67-0314-0010, NR 044-422, and by the National Science Foundation under grant GJ32111.

ABSTRACT

A simple model of parallel computation is a single instruction stream controlling a multiple processor configuration. Programs for such computers entail a host of considerations absent from programs for a conventional sequential computer. This paper explores the main considerations in using such a computer, largely in terms of the Illiac IV. It deals with gross system characteristics and how they affect the suitability of various problem formulations, parallel program structures and data representations, and coding strategies and techniques. The paper is self-contained in that it does not require any previous knowledge of the Illiac; it should be of interest both to the general computing community as a survey of practical aspects of parallel computation and to those actually contemplating using the Illiac.

INTRODUCTION

One of the simplest models of parallel computation is a single instruction stream controlling a multiple processor configuration. Programs for such a computer entail a host of considerations absent from programs for a conventional sequential computer. This paper contains some of the ideas that anyone who is formulating a problem, designing a program, or writing code for such a parallel computer should consider. They are drawn from the author's own experience and from conversations and published reports of many people who have used a particular realization of this architecture, the Illiac IV. Using a real computer to focus the discussion provides a concrete basis for these observations, but each problem is considered both as to how it affects programming the Illiac and as to why it is probably endemic to any realization of this parallel architecture. As a result this paper should be of use to scientists interested in programming the Illiac to solve their computer modeled problems. And it should also be of interest to the general computing community since it discusses the considerations involved in programming a single instruction stream multiple-processor computer and the various techniques that have evolved to exploit the power of a particular parallel machine within the restrictions of its architecture.

The three chapters display a progression from global considerations toward local and more specialized issues. The first chapter deals with system characteristics and how they affect the suitability of various problem formulations for efficient implementation on the Illiac. It is intended to give the reader some feeling for what parallel computation on the Illiac is really like (at least as it is practiced in 1975). The second chapter takes up the issue of parallel program structure and data representations. As these problems are only recently being understood in the sequential computer context, it is probably not surprising that this chapter only sketches some options which are available and indicates what appear to be the main issues involved. The final chapter is devoted to a discussion of coding approaches and techniques. Stopping just short of the gory details, the final chapter gives a careful description of how to deal with several common program structures in light of the Illiac's architecture; more than anything else these topics should give the reader a clear idea of the power and limitations of the machine, and of a single instruction stream multiprocessor in general.

There are two appendices. The first consists of a discussion of selected aspects of three projects which were designed to use the Illiac. The second is a selected and annotated bibliography on parallel computation in general and on the Illiac in particular.

A note on nomenclature is appropriate at this point. The usual description of the Illiac is an array processor. The rationale for this is that the processing elements are not complete processors since they do not decode their own instructions but rely on a central processor for direction. On the other hand, the ability to calculate the location of operands, togther with one insruction (the symmetric difference, for example), is suffcient for general computing, i.e. "is equivalent to a Turing machine." Thus each processing element is essentially a computer in its own right (provided the central processor repeatedly issued a symmetric dfference instruction). Both points of view are valuable; in some contexts one seems more appropriate than the other. This paper tends to use the term multiprocessor for the Illiac, if for no other reason than its pedagogical shock value.

Finally, there are a number of places where timing information is used in analyzing various aspects of the Illiac. These reflect the state of the machine during the fall of 1975. Since a number of enhancements to the hardware are planned, and even more are possible, they sould be viewed as conservative, even obsolete.

I. SYSTEM CONFIGURATION AND PROBLEM FORMULATION

The gross features of the Illiac to a large extent determine what forms of computation can be executed efficiently on this type of architecture. The extent to which a problem can be formulated so that the computation it requires fits one of these forms (or a combination of these forms) will determine the degree of utilization of the computer that can be expected. There are two levels of any proposed calculation over a large data base on the Illiac that must be considered. The first involves the global interactions of the data base which determine the data movement between a large rotating mass storage device and a smaller buffer memory where the actual computation is performed.

Although this brief discussion is couched entirely in terms of the Illiac, it is quite likely that characteristics associated with a heirarchical storage will always occur in scientific computing dealing with large data bases. The reason for this may be due to technology: the physical limit of the size of random access memories is less than the information requirements for computational models; or due to economics: semiconductor memories are more expensive per bit than disk memories; or due to algorithmic requirements: most large scale computations can be formulated so that each step requires random access to a relatively small section of the data, and the entire data base can be partitioned into such sections. In sequential computations, the global considerations are well known, since scientific computations over a large data base can easily exceed the primary storage capacity of even the largest current technology, standard architecture computers. The techniques developed for a sequential computer to obviate this problem are relevant to the Illiac and to parallel computers in general and are included here mainly as a convenience for the reader.

The second level considerations involve local interactions of data within the buffer memory. As such it is new with the Illiac-type architecture, and requires some care to keep from significantly degrading the rate of computation. This arises from the fact that each memory module has an associated processing unit. Thus the more operands for a given instruction which lie in different memory modules, the fewer the total number of instructions which will be required to process all data. This is in contradistinction to a sequential computer which requires one instruction (memory fetch) for each datum in memory. Thus the random access memory of an array computer is more a special type of pseudo - random access memory when computer efficiency is considered. Again this is inherent to a multi-processor (single instruction stream or not) since the problem is basically transferring information from the memory to the appropriate processing element. The idea is essentially to produce a data structure such that most processors are "close" to the information they need, or such that the necessary information can be distributed to the processors quickly. On the Illiac, the decision was made to put the mechanics of interprocessor communication (or processor access of non-local memory) explicitly under central program control. This was possible since the single instruction stream dictates a synchronous processor configuration, and one result is a simplification in the necessary hardware switching mechanism. Another result is a transfer of the hardware complexity up to the software level.

I.A. System Configuration

The Illiac is a large disk memory connected to a multiple-unit arithmetic facility with a semiconductor buffer memory. The disk is a fixed-head physically rotating device with a capacity of about sixteen million 64-bit words with a 40 millisecond rotation period and a maximum data transfer rate of about 15 million words per second to the buffer memory. The buffer memory consists of sixty-four modules, each module having 2048 words. Data transfers to or from the buffer are program initiated and are performed in blocks of 1024 words; the same contiguous locations in each module are affected. The transfer time for 1024 words is approximately sixty-six microseconds, or to refresh half the buffer memory takes a transfer time of 4.2 milliseconds.

Each buffer memory module has its own processing element--roughly the equivalent of a conventional computer's arithmetic unit. These execute the same instruction on data located in their own module. Information may be transferred among the modules one word per module in a carrousel fashion; for speed, special data paths make the time required for this routing less than linear in the number of modules by-passed. It is also possible to disable processing elements either under a central control or under conditions based on the contents of respective memory modules.

As a rule of thumb a general floating point operation (producing up to sixty-four results) takes about 1.7 microseconds. Typical bookkeeping operations by the processing elements take about 1.2 microseconds, and bookkeeping operations performed by the central control unit take about 0.7 microseconds.

The relative occurrences of each type of instruction in the calculation of short, simple arithmetic expressions that occur in inner-loop calculations is roughly twenty to thirty-five percent arithmetic, forty to sixty per cent processor bookkeeping, and twenty to thirty per cent central control bookkeeping. As for execution time, the ratios are approximately forty to fifty to ten, respectively. These figures do not include program control logic (i.e. loop counter updates and tests) which are usually weighted toward central control unit instructions, nor do they include arithmetic control logic (e.g. if x(i)< epsilon then x(i) := 0) which increases the amount of processor control; but they do include the address calculation overhead due to indexed arrays. The numbers to some extent reflect the particular instruction repertoire of the Illiac and the hardware implementation, but they give an indication of how much time and effort is spent in this form of parallel computation making sure that the correct data is being supplied to the appropriate processors.

Using these figures to estimate the computation constraints for disk-to-buffer data transfer to balance computation, about two hundred to two hundred fifty arithmetic operations per unknown should result in computation time equalling transfer time. This assumes one page (1024 words) is transferred at a time, a twenty millisecond average rotational delay, and a high concentration of arithmetic operations as occurs in evaluating arithmetic expressions. This last assumption is grossly unrealistic for general program behavior, of course, since general programs will have more control overhead per arithmetic operation. Thus in general fewer arithmetic operations per unknown will suffice to balance I/O with computation time. If thirty-two pages (one fourth of the buffer memory) are transferred at a time, the constraint drops to an average of seven to nine arithmetic operations per operand. If the data has been arranged on the disk so that the rotational delay is four milliseconds (one tenth of a rotation) then the arithmetics per unknown for computation time to equal I/O time is about forty to fifty for one page transfers and around two for thirty-two page transfers. The point of these numbers is to indicate the need to match buffer size and disk placement to the computational requirements of a program.

I.B. Problem Formulations: Global Considerations

What type of problem formulations lend themselves to an hierarchical memory structure? A review of some forms of computation which can efficiently be implemented on a disk-based memory is relevant at this point. Clearly for large data bases which cannot be held in the buffer memory for the entire computation it will be necessary to bring blocks of data into the buffer for processing. Thus the computation may be viewed as data flowing from the disk into the buffer memory for processing and then returning to the disk for storage until needed again.

The most obvious form of computation suited to this environment would be one in which each block of data on the disk could be treated independent from the others. This suggests that any information which is transmitted among the blocks be done in an indirect fashion through tables that reside permanently in the buffer memory. A simplified example will clarify matters.

Suppose that a simulation is to follow a group of particles moving about in some region and the region is divided into many areas. Suppose further that the way a particle moves depends only upon quantities local to it, plus a knowledge of how many particles are in each area of the region. Then the particles may be grouped arbitrarily into blocks on the disk and the particle interaction can be handled by a table in the buffer indicating how many particles are in each area. Each pass through the data will simulate one time step. During each pass information pertaining to each particle is updated and also information is gathered as to how the table of particle concentration is to be modified. Between passes over the disk-based data the table in core is updated.

A second general form of computation which lends itself to the constraints of this system would be one for which the computation performed on each block depends only on blocks that have recently been brought into the buffer and have not yet been written out to the disk again. As an example, consider a simulation performed on a mesh covering the surface of a cylinder. Each node of the mesh is updated according to the values of its neighbors during the previous time step. Suppose all the information for the nodes on the top of the cylinder fit into a block, and that the nodes on each circle of the mesh around the cylinder fit into a block also. Then the computation may proceed as follows. First the blocks containing information about the top nodes and the nodes in the ring nearest the top are read into the buffer. The top nodes are updated and the second ring is read into the buffer. Next the nodes on the first ring are updated, the updated nodes on the top are written to the disk, and the nodes of the third ring are brought into the buffer. This pattern continues until all nodes have been updated.

Notice that in this form of computation it is possible that more than one computational pass may be performed with each physical pass through the data. This can be illustrated in the preceding example. After the first ring has been updated, the values for the top nodes for the next time step may be calculated. After the third ring is updated, the second ring may be advanced another time step if the necessary information for this second time step (the once updated first and third rings) remains in the buffer. In general, suppose that six blocks can be held in the buffer for computational purposes. At the same moment ring i at time $t+1$, ring $i+1$ at time t and $t+1$ and ring $i+2$ at time t are in the buffer. Then if ring $i+3$ at time t is read in, the values of ring $i+2$ at time $t+1$ may be computed, then the values for ring $i+1$ at time $t+2$ may be computed and written out to the disk. Thus after ring $i+2$ is read in, the updated values for ring i at time $t+2$ may be written back onto the disk, and so two computational passes through the data will have been effected using only one physical pass through the data. The method can be generalized, limited only by the number of blocks that can be held in the buffer memory for computational purposes.

A third form of computation deals with several streams of data. If the computational dependence among the streams is such that the streams may be aligned and brought into the buffer from the disk so that all interactions involve only elements residing in the buffer, then this computation may be performed efficiently on a disk-based memory. The simplest example of this is a vector formulation of computation, where all computing may be expressed as component-wise combination of very long vectors. Here the long vectors on the disk are partitioned and appropriate segments from each vector which enter into the combination are brought into the buffer for processing.

It should be obvious that these three forms of computation lend themselves well to efficient computation on the Illiac or on any disk based computer with a suitable bandwidth between disk and core. Suitability depends, of course, on the computation required per data element. The greater the amount of processing required per element, the lower the threshold for the bandwidth to be considered acceptable. The degree to which data dependencies depart from these three forms will determine the difficulty in managing the data flow from the disk to the buffer. And the greater the difficulty, the more bookkeeping overhead will be neeeded to keep track of the added complexity.

I.C. Problem Formulation: Local Considerations

Computations are performed on data while it resides in the buffer; here the partitioned structure of the Illiac memory is the main consideration. The reason for this is that each memory module has its own arithmetic unit, hence in order to utilize as many of these units as possible, a premium is put on the alignment and distribution of operands entering into a parallel computation. There are several programming techniques to mitigate the effects of an unfortunate data structure, and these are discussed in chapter three, but the best technique is to minimize the difficulty at the time of the problem formulation.

There are two conceptually different ways of viewing the arithmetic facility. The first is that each address generated by the central facility specifies a location in sixty-four different contexts: the semantic meaning attached to each memory location by the programmer may differ depending upon the particular context. For example, suppose that the following two computations are to be performed: A*X+B and C-D (the first may be preparing to update a variable, the second preparing to check if an iteration may be terminated). Assume that the contents of the memory is

		module 0	module 1
location	#0	A	D
	#1	B	C
	#2	X	-1

Then the sequence of instructions

```
Load       0   (load the accumulator from location 0)
Multiply   2   (multiply accumulator by location 2)
Add        1   (add contents of location 1 to accumulator)
```

will leave the first result in the accumulator of memory module 0 and the second in the accumulator of memory module 1.

An example of where this technique might be useful may clarify matters. In some models which simulate the earth's weather, the ground condition can be in one of a few states (typically about six). The calculations of several variables, such as the heat capacity of a surface node, vary in their complexity according to the state of the ground at that node, but have roughly the same form for all states. A sequential code branches according to the ground state to separate sections of code, one for each state. An inefficient parallel code will turn on only those PE's processing surface nodes in a given state, and do this in turn for each possible state. An efficient parallel code will initialize certain variables (to values which appear as constants in the various cases of the scalar code) and then process all nodes with the same instruction sequence using these variables to encode the effect of the ground condition on the computation.

The second way of viewing the arithmetic facility is that each instruction initiates a computation on groups of homogeneous data. For example, suppose a vector A has A(0) in location 0 of module 8, A(1) in location 0 of module 9, and in general A(i) in location i/64 of module number (i+8) mod 64. Suppose a vector B has B(i) in location i/64 + 12 of module number i mod 64. Then to add the first sixty-four elements of A to the first sixty-four elements of B--that is to form A(i)+B(i) for i=0,1,...,63--the following instructions

```
If module number > 8 then index register := 0
                       else index register := 1;
Load R  0+index register    (load the first 64 elements of A into routing register R)
Route     8   (align vector A with corresponding  components of B by
                     routing a distance of 8 modules)
Load      R   (put contents of routing registers into the accumulator)
Add      12
```

produce the desired result: A(i)+B(i) is in the accumulator associated with memory module i.

These two points of view suggest two alternative forms of computation and data structures which will be efficient on the Illiac. The first is a homogeneous computation on data that is arranged analogously in different memory modules. An extension of this idea is to treat special cases, which in sequential programs would be handled by conditional tests and branches, by means of data rather than by enabling and disabling selected arithmetic units, the parallel equivalent of multiple branches. A practical application of this is the treatment of physical boundary points in a mesh with computations analogous to the interior nodes by the judicious addition of pseudo-grid points. This is discussed further in chapter two.

The second point of view suggests that homogeneous data (e.g. vectors) processed in a uniform fashion (e.g. by consecutive location within the vector) can also be executed efficiently on the Illiac. As a consequence, the greater the number of irregularities in the underlying model's structure which give rise to irregularities in the data structure's representation and treatment, the greater the penalty in programming complexity and execution time on the Illiac. Put quite simply, it may very well be that a fine, regular structure which requires a great deal of raw computational power is actually superior to an irregular structure (which might reflect the irregularities of the physical problem) which requires fewer arithmetic operations but considerably more detailed bookkeeping. This shift towards brute force is much more pronounced in the Illiac than in a sequential computer for the following reasons. On a sequential computer, each arithmetic operation, even on a regular structure, requires some bookkeeping, while on the Illiac, most of the comparable bookkeeping is implicit in the vector operations, and whatever explicit bookkeeping is done can be amortized over many operations (up to sixty-four). And secondly, bookkeeping operations on the Illiac (even setting a bit in a register) are much, much more expensive than the per-operand execution time when many arithmetic units are utilized.

It should be noted that most problems assumed to be ammenable to parallel computation will have some sections which are best formulated in terms of a single instruction stream directing computations having different meanings in different PE memories, while other sections are most efficiently formulated as vector operations. One of the reasons for this lies in the character of much of scientific computing. These problems are often formulated as solving sets of partial differential equations over time, and explicit finite difference methods make a vector formulation attractive in stepping through time. But the functions which determine the coefficients entering into the equations are often based on incompletely understood physical models, statistical approximations to subscale phenomena, or empirical measurements; whatever the reason, the result is usually an approximation based on a few parameters and several disjoint cases. Thus the calculation of those coefficients is frequently formulated using the first method.

I.D. Summary

In summary, as far as problem formulation is concerned, the major characteristics of the Illiac are a large rotating device for its primary memory and a 64-way partitioned memory buffer where arithmetic computation is performed. During the course of a computation, blocks of the data base are read into the buffer from the central memory; the extent to which these blocks are accessed in a regular fashion, and the amount of computation that can be performed per block read into the buffer, will determine the overall efficiency of an Illiac program. Three forms of data interdependencies among the blocks were discussed; each should form the basis of an efficient utilization of the disk memory.

If the data base is so poorly designed and arranged on the disk that most computation time is spent shuffling data between the disk and the buffer memory, then any arrangement of data within a block will suffice. But assuming that a problem is well formulated for the Illiac with respect to gross data interdependencies (or that the entire data base can be contained within the memory buffer), then the format of data within the blocks can be of crucial importance. This is the result of each memory module having its own arithmetic unit, the capability of operating on data within its own module or on data that has been uniformly routed among the modules. Since the same instruction is executed in each module at a given time, the form of each computation to be performed in parallel must be roughly the same; however, the actual meaning and relative location of each operand may vary from module to module. Also, since each operand in a route instruction must move the same distance, operations on data structures must be able to exploit uniform accesses across memory modules. Both local considerations point to homogeneous data structures replicated across memory modules. The remainder of this paper is essentially an elaboration of dealing with the local data considerations -- the interaction of a single instruction stream with a partitioned local memory.

II. PROGRAM AND DATA STRUCTURE

The major program design considerations on the Illiac arise from the restrictions of a single instruction stream and a partitioned memory. The effects of the instruction stream are more noticable in the program control structure than the effects of the memory on the data representations, but the two isuess are intimately connected. It is possible, however, to divorce the discussion of control structures from the problems of data representations, and at the level of program design such a separation is desirable. The single instruction stream is particularly attractive in parallel computation both from its conceptual simplicity and for the implementation advantages. For large scientific computations on large machines, neither advantage is to be underestimated. The same could be said for the partitioned memory, since in the Illiac the effect of the partitioning is manifest in the program only because the processing power has been distributed close to the data rather than separated into a separate processing facility as in more conventional designs and because the inter-processor routing necessitated by the distribution is under explicit program control.

Two overriding principles are relevant in the context of a multi-processor. The first is that generally the more processing elements doing useful work the better. (Of course, the caveat is that a reformulation of the problem might use fewer arithmetic units for a shorter total elapsed time and hence be better while making less use of the full parallelism of the machine.) For a single instruction stream this means that more arithmetic units should have some use for the current instruction being decoded by the central control unit; two techniques for achieving this are time-sharing the instruction stream and associative processing.

The other principle in multi-processors is to insure that the right information is available to the right processing element at the right time. There are two aspects to this principle. The first is the question as to whether it is better to recompute information needed at a particular site or to calculate the location of the stored information and retrieve it. This question is much more complex for a multi-processor than for a single processor computer with a random-access memory (or random-access and rotating storage configuration). In the multi-processor it is usually a question of an indeterminant number of processing elements, say n, calculating the information they need versus their calculating the location of the needed information (probably in fewer than n memory modules) and (if required by the architecture as in the Illiac) the various routing required for the information to reach the desired destination before finally getting the stored information to the appropriate processing element. Some programming techniques useful in implementing such a table look-up on the Illiac are discussed in chapter three.

The other aspect of the principle of aligning data to processing elements involves reducing the overhead associated with bookkeeping. In some cases the explicit bookkeeping can be reduced by judiciously inserting dummy data in the data representations stored in local processor memories and thereby forcing the desired alignment. Thus with the expense of some superfluous computation on this dummy data the control overhead of aligning data for computation can be greatly reduced. Two examples of this phenomenon are discussed below.

One last idea should be mentioned in passing. In sequential computing the idea of a dynamically varying data structure (e.g. a balanced binary tree) arises in an attempt to minimize the time spent traversing a linked data structure. In the Illiac an analogous situation arises when trying to minimize the time spent routing information among processing elements. The difference is that instead of changing pointers (and only logically changing the representation), portions of the structure actually change modules (and hence the site of the physical representation changes).

II.A. Instruction Stream and Program Organization

If each memory module is viewed as a separate context in which computation is performed, then the problem arises as to which context will control the instruction stream so that meaningful computation can proceed in its memory. In a multi-programmed sequential computer the same problem is solved by time-sharing the single instruction stream; the only difference is that in a sequential computer the programs running in different contexts are largely or completely independent, whereas in the Illiac there is likely to be rather strong and relatively frequent interaction among the different contexts, otherwise the question would arise as to why many smaller, independent machines were not used instead of a large multi-processor.

All this means is that the level of time-sharing (or level of coordination) is likely to be much finer in the Illiac -- in units of one to several instructions -- and the criteria for scheduling which group of instructions is to be executed next will be based either on a priori estimates as to which sequences are likely to keep the maximum number of processing elements active or based on some form of polling to determine which instructions will in fact be executed by the maximum number of elements.

The other approach to the problem is to have not one but several contexts within each memory module in the hope that when any given group of instructions is broadcast from the central unit, it will be applicable to at least one context in each module.

An example should clarify these ideas. Suppose that the main body of an algorithm consists of repeated executions of

If A then B else C.

This form of computation occurs, for example, in a tree searching algorithm where A decides whether to continue the search down a branch (i.e. execute B) or to backtrack (i.e. execute C); here every memory module contains the information needed to search a sub-tree during the computation. The problem with this form of computation on the Illiac is that in some contexts (memory modules) the execution of A will indicate that B is to be performed next, while in other contexts it will indicate that C is to be performed next. The way this is handled is to turn off all processing elements which want to execute C while the instructions for B are issued, and then turn off all processing elements which have executed B while the instructions for C are issued. Thus the code is equivalent to repeated executions of the sequence

A; If bool then B; If not bool then C

where bool is a boolean quantity set at the end of executing A.

However, after B is executed, rather than issue the instructions for C, A could be executed with the expectation that more processing elements will then be utilized when the instructions for C are issued. Thus the loop of the algorithm becomes

If bool then (B;A); If not bool then (C;A)

which requires that A be executed before the loop is entered to initialize the value of bool correctly.

Does this modification improve the processor utilization significantly? For general formulas to estimate the processor utilization in this example, let p be the probability that bool=true after the execution of A and let a be the time to execute A, b the time to execute B and c the time to execute C. Then the per centage of procesor utilization under the first scheme is

$$\frac{a+pb+(1-p)c}{a+b+c} .$$

For the second scheme assume that a steady state has been reached, that is in $N=p/(1-p+p*p)$ of the modules bool=true at the start of the loop (and hence after each pass through the loop). Then the estimated processor utilization for the second version is

$$\frac{a+Nb+(1-pN)c}{2a+b+c} .$$

We present a few examples. If $p=1/2$ then $a=b=c$ means 67% processor utilization for the first scheme versus 58% for the second and the first is clearly better. If the ratios of a:b:c are 1:1:4 then the utilizations are 58% and 62% , or about the same. For ratios of 1:4:4 the result is 56% to 63% and the second is clearly better. Now assuming $p=1/3$ the above three cases are 67% vs. 57%, 67% vs. 70%, and 56% vs. 61%.

Another approach is to keep track of how many contexts are in a state where their processing elements would be enabled if the code for A were broadcast, and the same for B and C. Then if the number which would execute A were greater than a specified number, the code for A would be broadcast, otherwise the number of processing elements which would execute the code for B is checked, and so forth. If the demands for codes A, B and C are all insufficient, then the threshold is lowered. Alternatively, the code for which there is the greatest demand could be chosen for execution whenever there is a choice.

The next modification is to have not one context in each memory module, but several independent ones. The assumption is that most of the time at least one of the contexts in each memory module will be in a state that can utilize whatever sequence of instructions is being broadcast by the central control unit. With three different states, as in the above example, this is fairly easy to insure with a relatively small number of contexts per module. However with a physical model which may have many logical states for any given context, this may not be possible given the small memory size of each module. It is in this case that the form of the computation becomes relevant, as discussed in the first chapter.

Since it is only the instruction stream that must be shared by the different contexts and not the semantics attached to how the instruction is modifying the context in which it is executed, distinct physical states should be formulated so that they can utilize the same instruction stream, the semantic differences being encoded in data variables. This is in contrast to the practice in sequential computing where computational differences reflecting the physical/semantic differences are imbedded in the instruction stream itself by means of branches. This difference from considerations entering into the problem formulation for a sequential computer requires that more care be taken both in the parallel formulation and in the actual program design and coding (and especially in the documentation).

All of the above discussion implicitly assumed that the execution time for each group of instructions was either the same in each context or that the variance was insignificant, i.e. that there are no loops requiring widely varying numbers of executions. The problem of eliminating program branches can thus be conceptually solved by means of using associative programming: the state of a particular context explicitly determines what sequence of actions are to be executed in it. But sequences that are of varying length in varying contexts defeat the strategy of initiating only those sequences which maximize processor utilization, since the tendency is for the worst-case local behavior of a given context to become the expected global behavior of an ensemble of contexts executing the same loop, and hence most processors will be idle waiting for the worst-case to terminate. This suggests going to a finer scale of computational resolution. Again the same idea of associative programming can be employed, but it takes on the flavor of time-slicing the instruction stream in a round-robin fashion.

To return to the above example, suppose that each block of code is itself a loop. For example

```
A = While A1 do AL.
```

Then the conditional expression can be re-coded for parallel execution as

```
If a1 then (AL; a1:=A1);
If b1 then (BL; b1:=B1; a1:= not b1);
If c1 then (CL; c1:=C1; a1:= not c1)
```

where AL has been modified to set b1 or c1 as appropriate when the loop AL is to terminate.

Thus each processor gets a chance to execute one pass through the appropriate loop on each pass through the reconstructed program. The obvious advantages in this reformulation are that all processors are busy at least some of the time during the loop and that no processor has to wait for all other processors to leave a given state before it can proceed to execute in another state. The disadvantage is, aside from the additional overhead, that a processor will necessarily be inactive through at least one and usually two of the if statements in any pass through the three statements.

All of the above program formulations have treated the processing elements as executing in separate contexts. Thus the programs look very much like programs for sequential computers, except that exceptional cases are handled by data rather than branches and that locally (within any one context) the code looks rather inefficient to the sequentially oriented programmer. But the efficiency comes from utilizing relatively more processing elements; e.g. significantly more than twice as many active processing elements executing code that runs about half as fast as a previous formulation will show an increase in total efficiency and a decrease in total execution time despite the more wide-spread local inefficiencies!

II.B. Partitioned Memory and Data Structures

If the Illiac is programmed as a vector machine capable of combining vectors of lengths up to sixty-four in a given instruction, then the problems of the single instruction stream treated in the previous section disappear, since by definition vectors are combined using the same arithmetic operation between individual components. A vector approach presents two different problems, however; one is specific to the Illiac's architecture and the other is inherent in the vector formulation of a program.

Vector operations on the Illiac directly encounter the effects of a partitioned memory, since to operate in parallel on two vectors in the buffer, they must be aligned so that components to be combined are available to the same processing element. An example in chapter one illustrated a particular case. The problem of vector alignment is more of a programming and data structure implementation detail, and will not be treated in this chapter which is aimed more at the level of program design. It is sufficient to realize that it is a nuisance that must be dealt with at the lower level of programming.

What must be dealt with at the level of program design is the elimination of special cases (which are usually handled with only a very limited degree of parallelism) and the creation of vectors that are as long as possible (so the overhead of calculating routing and indexing for the vector alignment can be amortized over more parallel arithmetic operations). Both of these concerns can be met by using computational artifices at the program design level. The aim is to try to enhance the computational parallelism of a problem formulated for solution on a parallel computer.

The idea of treating special cases ("boundary conditions") by means of special data rather than by means of special instructions has already been discussed when the same instruction is to be executed in different contexts. The same technique can be applied to vector-oriented computation; the goal is to incorporate special cases in such a way as to make the vectors involved in the program longer. As an example, suppose that a vector u is updated by

```
u(i) := c1(i)*u(i) + c2(i)*[u(i-1)+u(i+1)+u(i-k)+u(i+k)]
```

for most values of i except for the k values of i when i is a multiple of 200 where the relevant computation is

```
u(i) := c3(i)*u(i) + c4(i)*[u(i-1)+u(i+k)+u(i-k)].
```

The entire computation can be put in the first form and all the original values of u updated in the same fashion if zeros are inserted into u between every 200 elements, the vector c1 altered by inserting zeros and elements of c3, and the vector c2 altered by inserting zeros and elements of c4.

This example arises from a computation performed on a rectangular mesh, with the exceptional components of u corresponding to points along one boundary of the mesh. The expansion of the vector u corresponds to introducing pseudo-grid points along the physical boundary of the grid so that the original boundary mesh points can be treated in the same fashion as interior grid points. Notice that this is nothing more than a computing device designed for program efficiency; the pseudo-grid points do not affect the underlying physical model, they merely affect the representation of this model. The results of the two implementations should be identical (they are formally identical; any computational problems that may arise are due to the calculation and addition of zero affecting the low order bits of the addend because of floating-point vagaries of the arithmetic unit, but this should be negligible in any event). The only significant difference between the two formulations should be that the second, involving the longer vectors, runs faster.

As another example, suppose the 18 by 18 matrix A can be partitioned into 3 by 3 blocks so that it has the form

```
XX....
XXX...
.XXX..
..XXX.
...XXX
....XX
```

where each X is a possibly different 3 by 3 sub-matrix and each dot (.) represents a block consisting of all zeros. Thus recording only the elements in the sixteen non-zero blocks of the matrix A uses 16*9=144 elements. On the other hand, all the non-zero elements lie in eleven diagonals of A, centered around the main diagonal; two diagonals are known to be about one-third zeros and two others are known to be two-thirds zeros. Storing by diagonals (including the 'padding' zeros) takes about 11*18=198 elements. For general matrices with this structure, storing the 3 by 3 blocks takes about 9/11 of the storage as storing by diagonals.

To multiply a vector x by the matrix A in the first scheme requires sixteen smaller matrix-vector multiplies (the sub-vectors of x of length 3 by the 3 by 3 sub-matrices of A) followed by summing appropriate results and combining them to form the final product vector Ax. In the second scheme with A stored by diagonals, the product Ax requires component-wise multiplication of eleven diagonals and x and then summing the results. The second scheme in general takes more scalar arithmetic operations and, at least theoretically, slightly more parallel arithmetic instructions. But the second scheme requires a less complex control logic to keep track of the sub-computations involved in forming the product.

The final consideration is the lay-out of the data, both in the buffer memory and on the disk. Here the amount of routing and control necessary to align the data for computation within the buffer, together with the complexity of arranging the data into blocks and deciding which blocks should be brought into the buffer, can be much greater for the first scheme than for the second. The real complexity of the first approach depends upon the degree of general vectorization: whether the blocks are treated as general components of the three block-diagonals of the matrix A or as general non-zero blocks of the matrix. A successful implementation of the first scheme is discussed in section 1 of the first appendix, which also discusses the use of the Illiac instruction stream for associtive processing.

The central idea to notice in this discussion (which is also the thesis of APL) is the degree of control that can be subsumed by adopting a vector approach or a generalized vector approach (where vectors are themselves composed of matrices). This is a reflection of the use of the implicit bookkeeping possible when the Illiac is used in a vector mode.

II.C Summary

The two sections of this chapter have attempted via program design to deal with the issues of implementing a problem which has been formulated for a parallel computer. The chief aspects of the architecture of the Illiac which affect the program implementation are the single instruction stream and the partitioned buffer memory. The single instruction stream enhances the desirability of a vector formulation, or of the uniform treatment of various contexts in the memory modules. To some extent either can be accomplished by programming techniques, provided that the program is originally formulated to admit the use of such approaches.

The success of a multiple-context formulation depends upon the product of the number of possible computational states active at any stage of the computation and the number of variables necessary for each state. A relatively small product means several independent contexts per memory module and this insures a high probability that at least one of the contexts can be active for whatever section of code is being broadcast at a given time. Two ways were indicated for reducing the number of states. The first involved a detailed knowledge of state transition so that several states could be combined (in the example, state A always followed state B so that the two could be combined into a new state, (B;A), and the same for (C;A), thus the number of different computational states was reduced from three original states to two). The other way to reduce the number of distinct computational states was to formulate the way different physical states were to be handled in a computationally similar form so that the same instruction stream, with different parameters, is relevant to each state (an example of this occurred in the weather simulation discussed chapter one).

The success of a vector approach depends upon the ratio of exceptional cases to general cases. When this ratio is small, the vectors are long and thus a high degree of parallelism can be utilized for long periods of time. Long vectors are the key to efficient utilization of parallel computers since the vector start-up time, the time required to calculate alignment procedures for the vectors, can be amortized over many arithmetic operations.

III. PROGRAMMING TECHNIQUES

Specific programming examples indicate the range of considerations that arise in programming the Illiac, or any single instruction stream -- multiple processor computer. The essential problems are keeping track of the data and arranging the computation to process the data efficiently. Whereas in the preceeding chapter efficiency was assumed to be synonymous with processor utilization, here the term will be used to indicate minimal overhead to manage data and computation. There are several sources of overhead that appear unique to this type of parallel computing. One is data management-- having to rearrange data to get it to the right processor by routing. Another is processor management--having to decide which processors need to be active or inactive during a particular instruction sequence.

Routing, or inter-processor communication, plays a key role in all of the examples, even though they are all very simple (and basic). This arises for two reasons. The first is the decision to put control over processor memory accessing in the instruction set of the computer rather than having a hardware implemented switch which resolves possible memory conflicts and routing problems. The second factor is that frequent inter-processor communication should be characteristic of programs on a computer designed specifically for such rapid data swapping; problems which need far less processor interaction are probably better suited to a less tightly coupled network of processing elements.

III.A. Row Sum

Given sixty-four numbers x(0), x(1), x(2), ... x(63) to sum, a sequential computer must execute sixty-three addition instructions. However, if more than one add can be performed in parallel, fewer addition instructions are needed. The minimum number needed is six. This can be achieved as follows. First x(i) := x(i)+x(i+1) for even i.

Next

x(i) := x(i)+x(i+2) for i=0 mod 4.

The third instruction performs

x(i) := x(i)+x(i+4) for i=0 mod 8.

In general, the nth equation is

x(i) := x(i)+x(i+k) for i=0 mod 2k, where k=2**(n-1).

The implementation on the Illiac uses the route instruction to effect the offset by k, using both the routing register and the accumulator. This example was used by D. J. Kuck in "Illiac IV Software and Application Programming," IEEE Trans. on Comp., vol. C-17, 1968, pp. 758-770. Although the mathematical description implies that fewer scalar addition operations are done at each subsequent parallel step, in the Illiac it turns out that it is more efficient to perform sixty-four additions at each parallel step than it is to select that subset of processing elements which is needed to perform the useful additions. Assuming that x(i) is in the accumulator of memory module i, the code looks like

```
N := 1
Loop:
Store R  (put contents of accumulator in R)
Route N  (route the R registers a distance of N)
Add   R  (Add contents of R to accumulator)
N := N+N
If N < 33 then goto Loop
```

which terminates with the sum in every accumulator.

If more than sixty-four numbers are to be summed, say k numbers spread evenly among the memory modules, then k/64 additions are performed in each module before the row sum is computed. Obviousy the technique is not limited to addition but may be applied to any associative binary operator, for example finding the maximum.

There are two points which should be noted. The first is that a parallel algorithm can require much fewer instructions and take less time than a sequential formulation provided that the problem is formulated in a fashion amenable to parallel computation (here an associative operator was necessary) and that the data structure allows the parallel algorithm to proceed in parallel (if the summands were all in the same memory module, no parallelism would be possible without spending more time rearranging the data for a parallel algorithm than solving the problem sequentially on the arrangement as given). The second point to notice is that most of the executed additions are irrelevant to the parallel computation (only 63 of the executed 384 additions are necessary), yet no penalty was paid in increased execution time.

III.B. Routing Various Distances

The route instruction has the two attributes that the contents of all routing registers move, and they are all transfered the same distance. Thus when the desired goal is for elements in the routing registers to be routed varying distances, more than one instruction is necessary, but hopefully fewer than sixty-four will suffice. A particularly elegant solution to this problem is given in SAV-IV: A Three Dimensional Monte Carlo Radiation Penetration Code for the Illiac IV by Troubetzkoy, Kalos and Steinberg, DNA 3303F, 1973.

The central idea is that each element has a distance that it is supposed to move. The first step is to move elements whose distance has a one in its binary expansion, next those whose distance has a two in its binary expansion, then those with four, and so on. For example, an element which must move a distance of twenty-one would be moved one on the first instruction, four on the third instruction, and sixteen on the fifth instruction, since 1+4+16=21. It would be held in the local memory of some module during the instructions which move elements two, eight and thirty-two memory modules, and this suggests the major difficulty with the technique. It is possible that an element will arrive in a memory module and have to be stored when another element is already being kept there, and neither is supposed to move on the next instruction. This bunching phenomenon requires a temporary buffer in each module.

There are two alternatives in scheduling the routing instructions. The first is to move all elements which require a distance of one sometime in their move (this will take at most one route instruction), next all those which require a distance of two (this may take two route instructions because of the bunching effect), next all those needing four (which may take up to four route instructions), and so on. The other alternative is to sequence through the routing distances: moving a distance of one, next two, four, eight, sixteen, thirty-two, then two, four, ...

The above technique is especially helpful in implementing a table look-up on the Illiac. Notice, however, that it does require that the information know how far it must travel at the outset. To accomplish this, each processing element must be able to calculate in which memory module the desired information is stored and send a message to that module indicating the desired information. If the table is stored in an associative fashion, then there is no way to avoid a worst-case behavior requiring sixty-three route instructions: the keys to the information are passed among all the memory modules so the appropriate module can respond to the request.

The general principle that is being exploited here is breaking an action (here routing various distances) into constituent sub-actions, each having increased parallelism, in an attempt to increase the parallelism in solving the original problem and thereby to decrease the total execution time for the action (hopefully the advantage gained by the increased parallelism is not lost by the additional bookkeeping required to manage the sub-actions). The procedure can be more efficient when dealing with routing more than sixty-four elements since with several elements in each memory module the probability is increased that some element in each memory will be affected by each sub-action.

III.C. Higher Level Languages on the Illiac

The higher level languages available for programming the Illiac all possess the same fatal flaw: they make every effort to reveal all the vagaries of the machine to the programmer. By making the programmer painfully aware of the characteristics of the architecture, they mitigate the power of abstraction from machine details which is the greatest strength of a higher level language. The revelation is explicit in the data declarations where there is a new data type, a variable which consists of sixty-four scalars. (Needless to say it corresponds to one word in the identical location in each memory module.) The result of such an extension to languages otherwise much like Fortran or Algol is felt in the semantics of the language. Whenever one of these new data types occurs in an arithmetic expression, not one but sixty-four computations are performed in parallel. And there is a new monadic operator, the route operator, which when applied to the new data object causes the contents of its scalar positions to rotate. The assignment operator can even be modified when one of these new data types occurs on the left hand side of an assignment; a mode word determines which of the sixty-four constituent elements are to be altered by the assignment.

What has been forfeited by the languages must be recouped by programming techniques. Essentially what this means is that the program should be designed to run on a parallel computer with characteristics amenable to the problem, then this ideal computer simulated by the Illiac. The degree to which this logical machine resembles the actual computer will determine how efficiently the simulation can be done. This simulation may be explicitly programmed, or merely implicitly indicated by handling data structures larger than sixty-four elements in one dimension by sequencing through blocks of the structure, treating each block in parallel. Some techniques for simulating more amenable architectures for parallel problem formulations are discussed in the next sections.

Only after a program is running correctly should the question of coding efficiency be considered, since with the current level of understanding of parallel computation it requires some working prototype to be able to estimate accurately the disk-to-buffer data movement characteristics, the intra-buffer routing, the amount of bookkeeping, and the final degree of arithmetic parallelism of a given problem formulation and program design. There are enough cases of a priori optimization decisions that were simply wrong to cause any thoughtful person to question squandering resources on optimizing what may very well turn out to be irrelevant to the computing characteristics of a running program.

III.D. Simulating Processing Elements

One of the first difficulties that confronts a user of the Illiac is that there are sixty-four arithmetic units, no more and no less. To formulate a problem or write a program so that this exact number coincides with one of the dimensions of the major data structures may be too restrictive. On a conventional computer the problem of data structures larger than the number of arithmetic units is handled by having the single processing unit sequence over the structure one element at a time; on the Illiac the analogous operation is to sequence over the data structure sixty-four elements at a time.

In effect, what is being done is having the sixty-four processing elements simulate a machine with more (or fewer) processing elements. If the simulation is explicit then the program is designed to run on a computer with N processors, where N is a natural parameter of the problem (for example, the number of grid points along one dimension of a covering mesh) and then in some fashion simulating the logical computer having N processing elements by the physical computer having sixty-four processing elements.

For such a simulation, each memory module has N/64 contexts, complete with accumulator and routing register. Then the logical code is executed N/64 times, one for each context in the local memory module, except whenever a routing instruction is encountered. The routing instruction is the only means of communication among the simulated contexts, and a special subroutine is needed to simulate the general route among the simulated contexts.

The main difficulty in simulating a route instruction arises from the fact that the number of memory modules simulated need not be a multiple of sixty-four, nor need the routing distance be a multiple of sixty-four. The result of this is that there will be one row of the physical memory that will require special consideration, the row which contains the element destined for the first simulated processing element. In general, some of the elements in this row will have to be routed one distance in order to be aligned with the physical memory modules which contain the first simulated memories, while the rest will have to be shifted a different distance in order to be aligned with the physical memories containing the last numbered simulated memories.

Of course, the full generality of simulating many processing elements is seldom needed in general scientific computing on the Illiac. But the problem does reflect the common difficulties encountered when dealing with a data structure whose representation must be folded along one dimension to fit into the buffer memory.

III.E. Vector Instructions

The instruction set of the Illiac allows the programmer to deal in one instruction with vectors having a maximum length of sixty-four, but these vectors must first be aligned before the components can be combined. Again the restriction to sixty-four for the vector length is an artificial constraint for a problem formulation, and it is easily circumvented by partitioning longer vectors into sub-vectors of length sixty-four and operating in turn on each sub-vector in parallel. This has the additional advantage in that the computation required to calculate the necessary routing to align the first component of the two vectors gives the distance necessary to route each component and need not be recalculated during the course of the vector calculation. The saving is proportionately greater as the lengths of the vectors increase.

One trivial problem with operating on vectors is that they may not start in the first memory module; this may be the result of an algorithm which performs x(i)+x(i+j) over i for different values of j, or the result of packing vectors into memory when the vector lengths are not multiples of sixty-four and no "gaps" occur between vectors. To illustrate the technique of dealing with vectors, suppose the first thirty elements of a vector are in memory location M in modules thirty-four through sixty-three, the next sixty-four elements are in memory location M+1 in modules zero through sixty-three, and so on. To operate on the first sixty-four elements in parallel, the index registers are used, having been set to 1 in processing elements zero through thirty-three and to zero in the rest. Then any indexed fetch from the memory will always retrieve sixty-four contiguous vector elements. Although there is a slight penalty for indexing in a memory fetch, the simplification of the control structure makes this price insignificant.

In calculating the vector addition a := b + c there are three alternatives in aligning the vectors. First b could be aligned with c, the sum computed and the result aligned with a for storage; second c could be aligned with b and the resulting sum aligned with a; or third both b and c could be aligned with a before the sum is calculated. In general two of the three vectors b, c and the sum b+c must be routed to accomplish the vector operation. There are cases when each of these three alternatives is the most efficient, and for long vectors the savings in routing time by choosing the most efficient can be significant.

There is a slight complication in calculating the best alignment strategy. This arises because the time it takes to route a distance d is not linear in d. For example, it takes just as long to route a distance of eight memory modules as to route one memory module; this results from a rectangular interconnection scheme to connect the routing registers.

The central idea in this section, as in the previous section, is in extending the hardware instructions via software to act on more data items with a single conceptual instruction. Once one begins to think in terms of a single instruction stream acting on multiple data streams, the extension is rather natural, even if the implementation is somewhat baroque due to the hardware design (or mis-design with respect to the routing operation -- the problems encountered and their circuitious solutions are akin to the difficulties in implementing via software multiple-precision arithmetic without having access to the overflow bit).

III.F. Data Structures

When a data structure is represented in the buffer memory, the effect of the partitioned memory is again felt, depending upon how the structure is to be accessed, since the processor in each memory module enhances the desirability of having interacting components in the same module, or in near-by modules. For example, consider the array x(i), i=0 to 639. One arrangement is to have x(i) in memory module i mod 64, so that there are ten elements in each module.

If the array x is to be used in a computation of the form

$f(x(i),x(i+1))$ for $0<i<638$,

then the above storage requires no fewer route instructions than the number of function evaluations. If, however, x(0) through x(9) are stored in module number 0, x(10) through x(19) in module number one, and in general x(i) in module i/10 mod 64, then there is only one route per ten function evaluations. Of course, if f involves much computation, the time needed for routing becomes insignificant in either case.

If the offset is t, then the routing distance in the first scheme is also t, whereas in the second it is either 0 and 1 for $1<t<9$, 1 and 2 for $11<t<19$, and so forth. Shorter routing distances generally take less time, and if t varies during the algorithm, the second scheme is definitely preferable, especialy as x becomes longer.

The point here is quite simple. The way in which a data structure is represented in the buffer memory can significantly affect the efficiency of any code which realizes the intended algorithm, depending upon whether the representation is designed to ease or to thwart the patterns of access to the structure. A poorly conceived implementation can ruin even the most carefully designed parallel problem formulation and algorithm construction.

III.G. Summary

The programming techniques discussed in this chapter illustrate two general principles: decmposing an algorithm into sub-actions which are highly parallel and decomposing a data structure into blocks which can be handled in parallel. The first is largely in response to the limitations of a single instruction stream, plus the realities of computation (which dictate what computations depend upon previous results) and of architecture (which controls how the information generated by one computation can be made available for another). If it is a sequential algorithm that is being decomposed, then any parallelism in the resulting sub-tasks will result in a speed-up in the reformulation. However, the additional time required to set up for each sub-task and to manage the sub-tasks may make the theoretical speed-up of academic interest only. This is especially true if there is a great deal of management necessary to go from one sub-task to another.

The second principle is largely in response to the opportunity afforded by each memory module having its own arithmetic unit, again tempered by the realities of computation and architecture. Here the accessing requirements of the intended structure will largely determine the suitability of various representations. And if no representation seems particularly well suited for the parallel algorithm, this is a prima facie case that the fault lies in the parallel algorithm being parallel in theory only and that a reformulation is in order.

CONCLUSION

There are certain verities of computing that, when explicitly stated, are rather banal but which have a tendency to be forgotten when attention shifts to a new and unfamiliar computing domain. The first is that every computer, being a physical realization of some architecture, has its limitations which, once understood, are traditionally renamed as its computing characteristics. On a multiple processor computer, the outstanding limitation/characteristic is the number of processors. Sequential computers are able to cope with operations on arbitrarily long vectors despite having only one arithmetic unit; the fact that the Illiac has only sixty-four arithmetic units limits only the amount of parallelism possible in a given instruction, it has no effect on limiting the size of a data structure (its higher level languages notwithstanding).

Another observation is that operations on large data bases must, by the definition of large, contend with the problems of multiple levels of storage and the management thereof. The numerical boundary for large may change from machine to machine, but the techniques of arranging computation in such an environment do not. The relative sizes of the memories, the bandwidth between them, and the amount and rate of local computation on blocks of the data base may determine the relative importance of the various strategies to minimize the multiple level problem, but the problem is formally independent of whether the actual computation is parallel or sequential. That the problem naturally arises in some form or other in parallel computation can be seen from the following argument. One of the factors which makes parallel computation attractive is doing relatively simple things that are largely independent; the point of parallelism is to decrease execution time, and the only way simple independent actions could take a long time on a sequential computer is for them to be performed over a large data base.

The third comment is that meaningful computation entails some overhead; the relevant tautology is that nothing of value is free. The difficulty is to decide how much is too much, and the problem is complicated by these two facts of multi-processor computation: not only must the information's location be calculated (as in a sequential computer) but also it must arrive at the right processor; and processors must be controlled so that when they can do useful work, they do, and when they cannot, they do not interfere with active computations. Some computations require less overhead management than others, which is another way of saying that some actions are better suited to one type of architecture than other actions are. It is improbable that any non-trivial program can consist entirely of low-overhead computations. The ideal program is one which spends most of its time doing what its computer is best suited for. Unfortunately life is not all that simple. For example, consider a problem which consists of two parts A and B. On a sequential computer A requires ninety per cent of the computation time; a parallel computer can execute A twenty times faster, but takes twice as long to perform B. The parallel computer solves the problem four times faster than the sequential processor, but spends eighty per cent of its time doing B!

This brings up the final point: any program must spend most of its time doing something; hopefully it spends it where the power of its computational formulation lies, or at least in doing what captures the essence of the problem (and sets it apart from computation which avoids this time consuming aspect). If this is not the case, then it is doubtful whether the program as formulated should be running on that particular type of computer.

ACKNOWLEDGMENTS

The author wishes to thank S. H. Fuller, A. K. Jones, and E. A. Rich, CMU, for their comments on an earlier draft of this paper. In addition, the author is greatly indebted to Ms. Rich for the final production of the report.

APPENDIX I: CASE STUDIES

The following are some selected observations on several programs which have been written for the Illiac. The intention is that they may shed some light on the nature of scientific computing that may be amenable to parallel computation. These case studies are important both for the problem formulation strategies, program design decisions and coding techniques.

1. Sparse matrix multiply

The following three paragraphs are a somewhat edited quotation from a report on three- dimensional stress wave simulation for the Illiac, authored by Gerald Frazier and Christian Petersen (DNA 3331F report by Systems, Science and Software),pages 48 and 49.

> The time stepping process for this problem consists of the calculation U=V+A*W for each time step. The first term V is a vector and its calculation involves vector operations which require no interaction among the Illiac PE's. As a result, it is easily computed in parallel. Similar operations are involved in the calculation of the vector W. The significant calculation is the multiplication of the vector W by the large sparse matrix A. This multiplication accounts for almost all of the computation time that is required to complete one numerical time step. A sophisticated but simple mechanism has been developed to perform the sparse matrix multiply in parallel. The non-zero terms of A lie in 3x3 submatrices of A, no more than 27 such submatrices in any row of A. These are arranged on disk so that when read into memory each arrives in the PE which contains the three elements of W which enter into the computation of the product of the submatrix of A and W. Furthermore, as successive terms of A are read from disk the matrix row numbers increase monotonically (but not necessarily sequentially). This is done so that the sparse matrix multiply can be completed in the order of ascending row number.
>
> The first submatrix to arrive in each PE from the disk is multiplied by the appropriate three components of the vector W and the results are accumulated

> in a buffer along with the row number identifier. This operation allows some PE's to work ahead on other row numbers. Since several rows may be processed simultaneously, a look-ahead buffer is maintained in each PE which contains both the elements and their row numbers. Since rows will continuously be completed as new ones are started, the buffer need only be large enough to contain the maximum number to be worked on at one time in any given PE. On the average, all of the multiplies for about 2.4 rows of the sparse matrix multiply are completed at a time.
>
> During the matrix multiply, a test is made to see if all contributions from the sparse matrix multiply are ready to be summed for the node numbered n. If all of the row numbers from the submatrix multiply are greater than n, then all contributions for n are calculated (all PE's are now working on contributions to higher node numbers). The contributions for n are then summed and added to the other terms to obtain the advanced nodal displacement U(n). This displacement vector is stored in PEk, where k=n mod 64. If the contributions from row n+1 are completed, then node n+1 is also advanced in time, otherwise the next submatrix multiply in line for each PE is performed. The parallel submatrix multiplies, row sums, and disk reads continue until all of the A matrix has been processed and all nodes have been advanced in time. The entire operation is repeated for each time step.

This ends the quotation from the text. Some points are worth mentioning here. First of all is the surprise that the matrix-vector product is not programmed as vector operations but rather as separate processes (in the terminology of chapter one, the Illiac is being used not as a vector processor, but as multiple processors, each working largely in its own "context"). The difference in this case is essentially 99 vector component-wise multiplies (of vectors of length 3N, where N is the number of mesh nodes) plus aligning and summing the 99 result vectors, versus 27N matrix-vector products (involving 3x3 matrices) plus aligning and summing the 27N vectors (of length 3). The vector formulation costs about 18% more storage -- the added padding of zeros is necessary for alignment purposes--plus the concomitant increase in arithmetics--the multiplications by the padding zeros. On the other hand, the vector formulation eliminates the control structure which tests to see when all information for updating each node has been assembled and can be combined. It also eliminate the buffer management for these intermediate results. The real subtlety of the problem lies in the aligning and summing involved in the two approaches, plus the possible necessity (based on small core memory) to partition long vectors.

The non-vector approach does lend itself to matrices which arise from arbitrarily connected grids. But the automatic grid generation used by this project generates grids which are unions of regions homeomorphic to a cubical lattice, hence the structure of the matrix A will have large blocks along its diagonal where the above vector approach will hold, and its off-diagonal blocks, most of which are identically zero, will have an analogous vectorizable structure.

2. A model for disaster

The Tensor code (Final Report of the Tensor/Illiac IV Project, ARPA Order 1839 (UCRL-51467) by Tad Kishi, 1973) is based on a grid which moves with the material; the solution at a grid point involves information from nine neighboring points. Here whatever regularity exists in the grid at the beginning of the simulation is rapidly destroyed over the iterations, so a vector formulation of the sparse matrix is inappropriate. The next question is, can an Illiac-type architecture, viewed as each processor working in its separate context but doing roughly the same thing, provide a suitable environment for such calculations? Or is this a formulation best suited for some other type of computer?

Unfortunately, the project gives no answer, since it was a complete failure. In fact, The charitable thing would be to forget this fiasco entirely, but since a computer is what it appears to its users to be, it is important to consider this project, if only as a study in cognative psychology.

> The project was essentially doomed by its charter. "Bound by the primary requirement to reconfigure an existing production code, the development of effective parallel processing methods for the Illiac computer system has been an exceedingly difficult one. It could not have been accomplished by a simple translation of the existing Fortran code to a comparable language for the Illiac. The Fortran listing of the Tensor code is a poor substitute for documentation. It is next to impossible to understand the Tensor code or to derive effective algorithms for parallel processing from a code that was programmed in assembly language for a conventional computer and then brute force converted to Fortran. The task has only been accomplished by reformulating and reexamining the basic finite difference equations. Unfortunately, neither consistent nor complete set of equations of the existing code was available and had to be redeprived [sic] by members on the ARPA Tensor project." (One can only wonder what the sequential code has actually been computing all this time). (p. 3)

To seal the project's fate, it was decided to code in an assembler language. The reasons given were that the higher level languages were undergoing development and hence a) did not generate reasonable object code (which is irrelevant; bad code can be selectively tuned) and b) their programming support was minimal at best. The result of this decision was predictable, "once a course of action was decided upon, it was literally embedded in 'cement.' Programming in assembly language left little or no flexibility in our code development." (pp. 3-4) Thus the conclusions drawn by this project were largely due to the propogation of poor early design decisions. A stunning example of this occurred when the program was restructured, proving "that skewing of data, which we originally believed to be essential for efficient boundary calculations, was immaterial. To reconsider the skewing of data at this point in our code development was next to impossible. This is the price one pays when a code of this complexity is programmed in assembly language." (p. 14) There was an even greater price: the code never ran. "Two simulations runs have been attempted in this configuration. The code has crashed in loop 1 in the k=0 boundary routine. The results have been evaluated, but there are no plans to continue debugging." (p. 15)

What were the perceived problems of programming this formulation on the Illiac? There were essentially three. First, "The inherent geometric structure of the 64-PE Illiac computer system imposes an artificial boundary (modulo 64) on the grid system and must be contended with throughout the program for an array not commensurate with this base." (p. 6) Second, "considerations of the boundary calculations ... required skewing as a fundamental requirement of the problem logistics for efficient PE usage. However, a given storage assignment for one phase of the calculation may not be suited for another part of the calculation." (p. 7) And finally third, "The calculational procedures of the slip lines for the Illiac array processors require extensive movement of data across the PE's in order to meet the nearest neighbor requirements for the nine-point difference scheme. This is the result of the change in the nearest neighbor relationship with time. Thus the values necessary for interpolation may be in some arbitrary assignment across the processing elements." (p. 63)

The first perceived problem is illusory; it is solved by logically programming in a system of N processing elements and then simulating N processors using 64 or fewer processors. As seen above, the second problem actually turned out to be a red herring, and probably a costly one at that. The third problem, which is the heart of the matter of whether this formulation can be effectively used on an Illiac-type computer, arises from assuming a fixed data structure; but if the grid moves with the physics of the process, it seems reasonable to entertain the notion that its representation moves with the computation of the algorithm; this may not solve the problem, but it might mitigate its presumed seriousness. Another possible approach would be to use a grid structure fine enough so that slip lines and any other physically interesting phenomenon could be derived from calculations performed on the fixed grid--this would be an example of using raw computational power in place of the potentially staggering overhead of bookkeeping and routing of information needed for a more sophisticated formulation. This solution may not be aesthetically pleasing, but it might be the best cost-effective method (or even the only technologically feasible method for very large models). Since the purpose of computing is insight, the only question is whether this insight should be derived directly from the mechanics of the algorithm or be inferred from the results of the calculation.

Notice that all three problems have a common thread: the vagaries of the programming language, in revealing all of the machine characteristics, has given the greedy programmer more than enough rope to hang himself in trying to pull the last bit of speed out of the machine. This is a very serious problem, since it distracts from the real issues. "Skewing and the pseudo 64-PE boundary are new experiences and add to the difficulties in visualizing parallel processes in the Illiac." (p. 7)

3. Monte Carlo Methods on the Illiac

The real problem in the Tensor code is the interaction among dynamically varying groups of nodes, and the attendant bookkeeping necessary to locate specific nodes or assemble the necessary information. Monte Carlo methods which are formulated so that interactions among constituent elements are implicit can effectively minimize this overhead problem, but at the expense of substituting an apparent "randomness" in the control-flow. That this substitution can be successful on the Illiac must certainly be one of the ironies of contemporary computing, since "conventional wisdom" had held that the single-instruction stream was the constraining factor to the efficient utilization of the Illiac, which does not obviously lend itself to branch-driven programs. (Conventional wisdom also ignored the impact of the memory structure on effective data utilization, which probably will be the constraining factor once more experience with the Illiac is reported.)

A successful Monte Carlo code for the Illiac is reported in SAM-IV: a three dimensional Monte Carlo radiation penetration code for the Illiac IV by E. S. Troubetzkoy, M. H. Kalos and H. Steinberg of Mathematical Applications Group, Inc, DNA 3303F, 1973. Of particular insterest is the mechanics used to implement a disorderly control flow (one which takes many different branches when executed successively of different data by a sequential computer).

> "The major difficulty with attempting to implement a Monte Carlo code ... on the Illiac lies in the intrinsic disorderly nature of Monte Carlo logic. ... The order and the nature of the physical events have little, if any, correlation from [particle to particle]. The naive approach of following 64 histories simultaneously is therefore not feasible as the parallelism breaks down almost immediately. Our approach is to initiate many histories in each PE, and hold all of them in abeyance until any calculation is required"--that is, until enough PE's have particles upon which the same calculation can be performed. (p. 10)

The basic idea here is reminiscent of the control mechanism in a production system, or Markov algorithm, where at least conceptually processes are activated in an associative manner whenever certain specified conditions in the data base arise. In the Monte Carlo program certain computations are performed whenever a certain amount of parallelism is possible.

Conclusions

A general statement of the philosophy underlying the successful programming strategy described in both sub-sections one and three would be: divide the problem formulation into as many independent steps as possible--steps which would have to be executed repeatedly on varying data by a sequential computer--and then at each point of the parallel computation, choose to execute that step which will utilize the greatest amount of parallelism. The ultimate success of any code seems to lie in the ability to minimize the overhead of bookkeeping, either implicitly (as in sub-section one where, for example, the computation required for a particular node is known to be completed when all PE's are working on computations involving higher numbered nodes) or explicitly (as in sub-section three where the formulation is in theory without any dynamically varying interrelationships among distinct components; that is, the aggrigate effects of interest can be viewed as data reduction which can be done without regard to order and in a cumulative fashion and hence lends itself well to homogeneous parallel processing).

One of the unifying characteristics of these three projects is their unwillingness to view the Illiac as a vector computer. This may be because of the small random access memory (implicit in the approach of sub-section one) or because of the short natural vector length (explicitly mentioned in sub-section two). Or it could be a (perhaps deserved) infatuation with a sequential program (as in sub-section three where, despite the dazzling programming techniques used to cope with the architecture of the Illiac, the code is essentially a parallel implementation of a sequential program). However, if one generalizes the notion of a vector operation from component-wise scalar operations to more complex operations on structured components, then these programs may be interpreted as attempts to simulate generalized vector computations.

APPENDIX II: BIBLIOGRAPHY

The following annotated biblography of papers dealing with algorithms for a general single instruction stream multiple data stream computer and for the Illiac in particular were selected from the open literature and hence should be readily accessible. The first four are interesting both for the particular application discussed and also for the underlying technques which are applicable to a much wider range of problems.

S.-C. Chen and D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," IEEE Trans. on Comp., vol. C-24, 1975, pp. 701-717. The paper contains an example of folding an algorithm designed for n PE machines to execute on an m PE machine where m<n.

H. Robert Downs, "Real-Time Algorithms and Data base Management on Illiac IV," IEEE Trans. on Comp., vol. C-22, 1973, pp. 773-777. Although couched in terms of radar tracking, the "real-time" constraint can arise from viewing the disk location as the external event driving the course of a computation. There are also a number of interesting ideas in managing a data base with special dynamic characteristics.

A. H. Sameh, "On Jacobi and Jacobi-like Algorithms for a Parallel Computer," Math. Comp., vol. 25, 1971, pp. 579-590. Many steps in the Jacobi method for reducing a matrix to diagonal form in solving a linear system of equations are independent and groups of them may be done in parallel (i.e. the computation is folded in time). The details of this in terms of the Illiac are presented.

H. S. Stone, "An Efficient Parallel Algrithm for the Solution of a Tridiagonal Linear System of Equations," J. of ACM, vol. 20, 1973, pp. 27-38. This paper presents an exposition of the technique known as recursive doubling in terms of solving first and second-order linear difference equations on the Illiac.

The following two papers are mostly of historical interest in describing the Illiac at various points in time.

G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, R. A. Stoker, "The Illiac IV Computer," IEEE Trans. on Comp., vol. C-17, 1968, pp. 746-757.

W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, D. L. Slotnick, "The Illiac IV System," Proc. IEEE, vol. 60, 1972, pp. 369-388.

Finally, two additional bibliographies in the field conclude this list.

Don Heller, "A Survey of Parallel Algorithms in Numerical Linear Algebra," Carnegie-Mellon University Department of Computer Science Technical Report, to be published.

W. G. Poole, Jr. and R. G. Voigt, "Numerical Algorithms for Parallel and Vector Computers: An Annotated Bibliography," Computing Reviews, vol. 15, 1974, pp. 379-388.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

3 8462 00613 0607

REPORT DOCUMENTATION PAGE		BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PROGRAMMING THE ILLIAC IV		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David K. Stevenson		8. CONTRACT OR GRANT NUMBER(s) N00014-67-0314-0010, NR 044-422
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE November 1975
		13. NUMBER OF PAGES 41
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
A simple model of parallel computation is a single instruction stream controlling a multiple processor configuration. Programs for such computers entail a host of considerations absent from programs for a conventional sequential computer. This paper explores the main considerations in using such a computer, largely in terms of the Illiac IV. It deals with gross system characteristics and how they affect the suitability of various problem formulations, parallel programs structures and data representations, and coding strategies and techniques. The paper is self-contained in that it does not require any previous knowledge of the Illiac; it should be of interest both to the general computing sommunity as a survey of practical aspects of parallel computation and to those actually contemplating using the [illegible]

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED